Certainly! Let's extend the previous implementation to include the `FasterRCNNWithCount` class and integrate it seamlessly into the training and inference pipeline. This enhanced model will not only perform object detection (identifying and localizing individual fish seeds) but also predict the total count of fish seeds in an image.

Overview

1. Phase 1: Train the model on images containing a single fish seed with bounding box annotations.
2. Phase 2: Fine-tune the model on a dataset with multiple fish seeds per image, where each image is labeled with the total count of fish seeds (without bounding boxes).

Table of Contents

Prerequisites

Ensure you have the following libraries installed. Install any missing libraries using `pip`:

```
pip install torch torchvision matplotlib pandas tqdm
```

- PyTorch: For model implementation and training.
- Torchvision: Provides access to pre-trained models and utilities for computer vision.
- Matplotlib: For visualization.
- Pandas: For handling CSV files.
- tqdm: For displaying progress bars.

Directory Structure

Organize your dataset as follows:

```
dataset/
├── phase1/
│   ├── images/
│   │   ├── img1.jpg
│   │   ├── img2.jpg
│   │   └── ...
│   └── annotations.json
├── phase2/
│   ├── images/
│   │   ├── img1.jpg
│   │   ├── img2.jpg
│   │   └── ...
│   └── counts.csv
```

- Phase 1:
    - Images: Stored in `dataset/phase1/images/`.
    - Annotations: `dataset/phase1/annotations.json` in COCO format with bounding boxes for single fish seed images.
- Phase 2:
    - Images: Stored in `dataset/phase2/images/`.
    - Counts: `dataset/phase2/counts.csv` containing image filenames and corresponding fish seed counts.

Example `counts.csv`:

```
filename,count
img1.jpg,5
img2.jpg,12
img3.jpg,7
...
```

Dataset Preparation

Phase 1: COCO Annotations

Ensure that your `annotations.json` for Phase 1 follows the COCO format. Here's a simplified example for images with single fish seeds:

```
{
    "images": [
        {
            "id": 1,
            "file_name": "img1.jpg",
            "width": 800,
            "height": 600
        },
        {
            "id": 2,
```

```
            "file_name": "img2.jpg",
            "width": 800,
            "height": 600
        },
        ...
    ],
    "annotations": [
        {
            "id": 1,
            "image_id": 1,
            "category_id": 1,
            "bbox": [100, 150, 50, 60],  // [x, y, width, height]
            "area": 3000,
            "iscrowd": 0
        },
        {
            "id": 2,
            "image_id": 2,
            "category_id": 1,
            "bbox": [200, 250, 45, 55],
            "area": 2475,
            "iscrowd": 0
        },
        ...
    ],
    "categories": [
        {
            "id": 1,
            "name": "fish_seed",
            "supercategory": "none"
        }
    ]
}
```

Phase 2: Counts CSV

For Phase 2, prepare a `counts.csv` file that maps each image to its corresponding fish seed count. This CSV will be used to correlate images with their counts during inference.

Implementation

Below is the comprehensive implementation, including the `FasterRCNNWithCount` model.

Imports and Utilities

```
import os
import json
import pandas as pd
import numpy as np
import torch
import torch.utils.data
from torch.utils.data import DataLoader, Dataset
import torchvision
from torchvision import transforms as T
```

```
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from collections import defaultdict
from tqdm import tqdm
import torch.nn as nn
```

Custom Dataset Classes

We'll define two custom dataset classes:

1. `FishSeedDetectionDataset` for Phase 1 (object detection with bounding boxes).
2. `FishSeedCountingDataset` for Phase 2 (counting without bounding boxes).

```
class FishSeedDetectionDataset(Dataset):
    def __init__(self, root, transforms=None):
        """
        Args:
            root (str): Root directory containing 'images/' and 'annotations.json'.
            transforms (callable, optional): Optional transform to be applied on a
sample.
        """
        self.root = root
        self.transforms = transforms

        # Load annotation file
        annotation_file = os.path.join(root, "annotations.json")
        with open(annotation_file) as f:
            self.coco = json.load(f)

        # Create mappings
        self.image_id_map = {img['file_name']: img['id'] for img in
self.coco['images']}
        self.annotations = defaultdict(list)
        for ann in self.coco['annotations']:
            # Associate annotations with file names
            image_info = next((img for img in self.coco['images'] if img['id'] ==
ann['image_id']), None)
            if image_info:
                self.annotations[image_info['file_name']].append(ann)

        # List of image filenames
        self.imgs = list(sorted(os.listdir(os.path.join(root, "images"))))

    def __len__(self):
        return len(self.imgs)

    def __getitem__(self, idx):
        # Load image
        img_path = os.path.join(self.root, "images", self.imgs[idx])
        img = Image.open(img_path).convert("RGB")

        # Load annotations
```

```
        ann = self.annotations[self.imgs[idx]]

        boxes = []
        labels = []
        areas = []
        iscrowd = []

        for obj in ann:
            xmin, ymin, width, height = obj['bbox']
            boxes.append([xmin, ymin, xmin + width, ymin + height])
            labels.append(1)  # 'fish_seed' category
            areas.append(obj['area'])
            iscrowd.append(obj['iscrowd'])

        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        labels = torch.as_tensor(labels, dtype=torch.int64)
        areas = torch.as_tensor(areas, dtype=torch.float32)
        iscrowd = torch.as_tensor(iscrowd, dtype=torch.int64)

        image_id = torch.tensor([self.coco['images'][idx]['id']])

        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["image_id"] = image_id
        target["area"] = areas
        target["iscrowd"] = iscrowd

        if self.transforms:
            img = self.transforms(img)

        return img, target

class FishSeedCountingDataset(Dataset):
    def __init__(self, root, counts_file, transforms=None):
        """
        Args:
            root (str): Root directory containing 'images/'.
            counts_file (str): Path to 'counts.csv'.
            transforms (callable, optional): Optional transform to be applied on a
sample.
        """
        self.root = root
        self.transforms = transforms
        self.counts_df = pd.read_csv(counts_file)
        self.imgs = list(self.counts_df['filename'])
        self.counts = list(self.counts_df['count'])

    def __len__(self):
        return len(self.imgs)

    def __getitem__(self, idx):
        # Load image
        img_path = os.path.join(self.root, "images", self.imgs[idx])
        img = Image.open(img_path).convert("RGB")
```

```
        count = self.counts[idx]

        if self.transforms:
            img = self.transforms(img)

        return img, count
```

Model Definition

We'll define a `FasterRCNNWithCount` class that extends `FasterRCNN` by adding a regression head to predict the total count of fish seeds in an image.

```
class FasterRCNNWithCount(nn.Module):
    def __init__(self, num_classes):
        super(FasterRCNNWithCount, self).__init__()
        # Initialize Faster R-CNN
        self.faster_rcnn = get_model(num_classes)
        self.num_classes = num_classes

        # Add a regression head for counting
        # We'll use average pooling on the backbone features and then a fully
connected layer
        # Alternatively, more sophisticated feature aggregation can be used
        self.count_regressor = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 1)  # Predicting the count
        )

    def forward(self, images, targets=None, counts=None):
        """
        Args:
            images: List of images tensors.
            targets: List of target dictionaries (for detection).
            counts: List or tensor of counts (for counting).
        Returns:
            If training:
                Combined loss dictionary
            Else:
                Detections
        """
        if self.training:
            # Detection losses
            loss_dict = self.faster_rcnn(images, targets)

            # Counting losses
            if counts is not None:
                # Extract features from the backbone
                features = self.faster_rcnn.backbone(images.tensors)
                # Suppose we take the 'avgpool' layer from ResNet
                # For ResNet50, the backbone returns higher-level features
                # We'll perform global average pooling on backbone features
                # and then pass through the regressor
                # Note: Adjust based on the actual backbone output
                # Here, assuming features are a dict with '0' key
```

```
                # which is standard for Faster R-CNN with ResNet backbone
                backbone_features = features['0']  # Shape: [batch_size, C, H, W]
                pooled_features = 
torchvision.ops.adaptive_avg_pool2d(backbone_features, (1, 1))
                pooled_features = pooled_features.view(pooled_features.size(0), -1)
# [batch_size, C]
                count_preds = self.count_regressor(pooled_features).squeeze(1)  # 
[batch_size]

                # Compute L1 Loss for counting
                count_targets = torch.tensor(counts, 
dtype=torch.float32).to(images.tensors.device)
                count_loss = nn.L1Loss()(count_preds, count_targets)

                # Combine losses
                total_loss = loss_dict['loss_classifier'] + loss_dict['loss_box_reg']
+ loss_dict['loss_objectness'] + loss_dict['loss_rpn_box_reg'] + count_loss
                loss_dict['loss_count'] = count_loss
                loss_dict['total_loss'] = total_loss

                return loss_dict
            else:
                return self.faster_rcnn(images, targets)
        else:
            # Inference
            detections = self.faster_rcnn(images)
            return detections
```

Note: The `FasterRCNNWithCount` class adds a counting regression head that takes aggregated features from the backbone and predicts the total count of fish seeds in the image. The loss during training combines the detection losses with the counting loss.

Model Setup Function

Define a function to initialize the Faster R-CNN model.

```
def get_model(num_classes):
    # Load a pre-trained Faster R-CNN model
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)

    # Get the number of input features for the classifier
    in_features = model.roi_heads.box_predictor.cls_score.in_features

    # Replace the pre-trained head with a new one
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    return model
```

Training Functions

We'll define two training functions:

1. `train_model` for Phase 1 (object detection only).
2. `train_model_with_count` for Phase 2 (object detection + counting).

Phase 1: Training Object Detection

```
def train_model(model, optimizer, data_loader, device, epoch):
    model.train()
    running_loss = 0.0
    for images, targets in tqdm(data_loader, desc=f"Epoch {epoch} Phase 1"):
        images = list(image.to(device) for image in images)
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        loss_dict = model(images, targets)

        # Total loss is the sum of individual losses
        losses = sum(loss for loss in loss_dict.values())

        running_loss += losses.item()

        optimizer.zero_grad()
        losses.backward()
        optimizer.step()

    avg_loss = running_loss / len(data_loader)
    print(f"Epoch {epoch} Phase 1 - Average Loss: {avg_loss:.4f}")
```

Phase 2: Training Object Detection and Counting

```
def train_model_with_count(model, optimizer, data_loader, device, epoch):
    model.train()
    running_loss = 0.0
    for images, targets, counts in tqdm(data_loader, desc=f"Epoch {epoch} Phase 2"):
        images = list(image.to(device) for image in images)
        # Prepare targets and counts
        batch_targets = []
        batch_counts = []
        for tgt, cnt in zip(targets, counts):
            if tgt is not None:
                batch_targets.append(tgt)
            if cnt is not None:
                batch_counts.append(cnt)

        # Detection targets
        detection_targets = [t.to(device) for t in batch_targets]

        # Forward pass
        loss_dict = model(images, targets=detection_targets, counts=batch_counts)

        # Total loss
        losses = loss_dict['total_loss']

        running_loss += losses.item()

        optimizer.zero_grad()
        losses.backward()
        optimizer.step()

    avg_loss = running_loss / len(data_loader)
    print(f"Epoch {epoch} Phase 2 - Average Loss: {avg_loss:.4f}")
```

Evaluation Functions

We'll define evaluation functions for both phases.

Phase 1: Evaluation for Object Detection

```
def evaluate_model(model, data_loader, device):
    model.eval()
    total_loss = 0.0
    with torch.no_grad():
        for images, targets in tqdm(data_loader, desc="Evaluating Phase 1"):
            images = list(image.to(device) for image in images)
            targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

            loss_dict = model(images, targets)
            losses = sum(loss for loss in loss_dict.values())
            total_loss += losses.item()
    avg_loss = total_loss / len(data_loader)
    print(f"Phase 1 Evaluation - Average Loss: {avg_loss:.4f}")
    return avg_loss
```

Phase 2: Evaluation for Counting

```
def count_fish_seeds(model, data_loader, device, threshold=0.5):
    model.eval()
    predicted_counts = []
    actual_counts = []
    with torch.no_grad():
        for images, counts in tqdm(data_loader, desc="Counting Phase 2"):
            images = list(image.to(device) for image in images)
            outputs = model.faster_rcnn(images)  # Only use detection for counting

            for output, count in zip(outputs, counts):
                detected = 0
                for score, label in zip(output['scores'], output['labels']):
                    if score > threshold and label == 1:
                        detected += 1
                predicted_counts.append(detected)
                actual_counts.append(count)

    predicted_counts = np.array(predicted_counts)
    actual_counts = np.array(actual_counts)

    mae = np.mean(np.abs(predicted_counts - actual_counts))
    rmse = np.sqrt(np.mean((predicted_counts - actual_counts) ** 2))

    print(f"Phase 2 Counting - MAE: {mae:.2f}, RMSE: {rmse:.2f}")
    return mae, rmse, predicted_counts, actual_counts
```

Visualization Functions

We'll define functions to visualize detections and counts.

```
def visualize_predictions_phase1(model, dataset, device, threshold=0.5,
num_samples=5):
    model.eval()
```

```
    for i in range(num_samples):
        img, target = dataset[i]
        with torch.no_grad():
            prediction = model([img.to(device)])[0]

        img_np = img.permute(1, 2, 0).cpu().numpy()

        fig, ax = plt.subplots(1)
        ax.imshow(img_np)
        true_count = len(target['boxes'])
        pred_count = 0
        for box, score, label in zip(prediction['boxes'], prediction['scores'],
prediction['labels']):
            if score > threshold and label == 1:
                pred_count += 1
                xmin, ymin, xmax, ymax = box.cpu()
                rect = patches.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
                                         linewidth=2, edgecolor='r',
facecolor='none')
                ax.add_patch(rect)
                ax.text(xmin, ymin, f"{score:.2f}", bbox=dict(facecolor='yellow',
alpha=0.5))

        plt.title(f"Ground Truth: {true_count}, Predicted: {pred_count}")
        plt.axis('off')
        plt.show()

def visualize_predictions_phase2(model, dataset, device, threshold=0.5,
num_samples=5):
    model.faster_rcnn.eval()
    for i in range(num_samples):
        img, count = dataset[i]
        img_input = img.unsqueeze(0).to(device)
        with torch.no_grad():
            prediction = model.faster_rcnn(img_input)[0]

        img_np = img.permute(1, 2, 0).cpu().numpy()

        fig, ax = plt.subplots(1)
        ax.imshow(img_np)
        pred_count = 0
        for box, score, label in zip(prediction['boxes'], prediction['scores'],
prediction['labels']):
            if score > threshold and label == 1:
                pred_count += 1
                xmin, ymin, xmax, ymax = box.cpu()
                rect = patches.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
                                         linewidth=2, edgecolor='r',
facecolor='none')
                ax.add_patch(rect)
                ax.text(xmin, ymin, f"{score:.2f}", bbox=dict(facecolor='yellow',
alpha=0.5))

        plt.title(f"Actual: {count}, Predicted: {pred_count}")
```

```
        plt.axis('off')
        plt.show()
```

Main Training and Inference Pipeline

We'll combine everything into the main function.

```
def main():
    # Paths to dataset phases
    phase1_root = 'dataset/phase1'  # Replace with your actual path
    phase2_root = 'dataset/phase2'  # Replace with your actual path
    phase2_counts_file = os.path.join(phase2_root, "counts.csv")

    # Define transforms
    transform = T.Compose([
        T.ToTensor(),
    ])

    # Create datasets
    dataset_phase1 = FishSeedDetectionDataset(phase1_root, transforms=transform)
    dataset_phase2 = FishSeedCountingDataset(phase2_root, phase2_counts_file,
transforms=transform)

    # Split Phase 1 into training and validation
    val_split = 0.1
    num_val = int(len(dataset_phase1) * val_split)
    num_train = len(dataset_phase1) - num_val
    dataset_phase1_train, dataset_phase1_val =
torch.utils.data.random_split(dataset_phase1, [num_train, num_val])

    # DataLoaders for Phase 1
    data_loader_phase1_train = DataLoader(dataset_phase1_train, batch_size=4,
shuffle=True, num_workers=4,
                                         collate_fn=lambda x: tuple(zip(*x)))
    data_loader_phase1_val = DataLoader(dataset_phase1_val, batch_size=4,
shuffle=False, num_workers=4,
                                       collate_fn=lambda x: tuple(zip(*x)))

    # DataLoader for Phase 2
    # For Phase 2, we need to provide images with counts. We'll create a combined
dataset.
    # To handle both datasets, we'll use a custom collate_fn
    class CombinedDataset(Dataset):
        def __init__(self, phase1_dataset, phase2_dataset):
            self.phase1_dataset = phase1_dataset
            self.phase2_dataset = phase2_dataset

        def __len__(self):
            return len(self.phase1_dataset) + len(self.phase2_dataset)

        def __getitem__(self, idx):
            if idx < len(self.phase1_dataset):
                img, target = self.phase1_dataset[idx]
                return img, target, None  # No count label
            else:
                img, count = self.phase2_dataset[idx - len(self.phase1_dataset)]
```

```
                return img, None, count  # No target annotations

    combined_dataset = CombinedDataset(dataset_phase1_train, dataset_phase2)
    data_loader_combined = DataLoader(combined_dataset, batch_size=4, shuffle=True,
num_workers=4,
                                      collate_fn=lambda x: tuple(zip(*x)))

    # Define the device
    device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
    print(f"Using device: {device}")

    # Define the number of classes (background + fish_seed)
    num_classes = 2

    # Initialize the model
    model = FasterRCNNWithCount(num_classes)
    model.to(device)

    # Define optimizer (commonly SGD for Faster R-CNN)
    params = [p for p in model.parameters() if p.requires_grad]
    optimizer = torch.optim.SGD(params, lr=0.005,
                                momentum=0.9, weight_decay=0.0005)

    # Learning rate scheduler
    lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                                   step_size=3,
                                                   gamma=0.1)

    num_epochs_phase1 = 5
    num_epochs_phase2 = 10

    # Phase 1: Train object detection
    print("Starting Phase 1 Training (Object Detection)...")
    for epoch in range(1, num_epochs_phase1 + 1):
        train_model(model.faster_rcnn, optimizer, data_loader_phase1_train, device,
epoch)
        evaluate_model(model.faster_rcnn, data_loader_phase1_val, device)
        lr_scheduler.step()

    print("Phase 1 Training Completed.\n")

    # Save the Phase 1 model
    torch.save(model.state_dict(), "fasterrcnn_fish_seed_phase1.pth")
    print("Phase 1 Model saved as fasterrcnn_fish_seed_phase1.pth")

    # Visualization for Phase 1
    visualize_predictions_phase1(model.faster_rcnn, dataset_phase1_val, device,
threshold=0.5, num_samples=3)

    # Phase 2: Fine-tune with counting
    print("Starting Phase 2 Training (Object Detection + Counting)...")
    for epoch in range(1, num_epochs_phase2 + 1):
        train_model_with_count(model, optimizer, data_loader_combined, device, epoch)
        lr_scheduler.step()
```

```
    print("Phase 2 Training Completed.\n")

    # Save the Phase 2 model
    torch.save(model.state_dict(), "fasterrcnn_fish_seed_phase2.pth")
    print("Phase 2 Model saved as fasterrcnn_fish_seed_phase2.pth")

    # Evaluation for Phase 2
    mae, rmse, predicted_counts, actual_counts = count_fish_seeds(model,
DataLoader(dataset_phase2, batch_size=4, shuffle=False, num_workers=4), device,
threshold=0.5)

    # Visualization for Phase 2
    visualize_predictions_phase2(model, dataset_phase2, device, threshold=0.5,
num_samples=3)

    # Optionally, plot predicted vs actual counts
    plt.figure(figsize=(8,6))
    plt.scatter(actual_counts, predicted_counts, alpha=0.6)
    plt.plot([actual_counts.min(), actual_counts.max()], [actual_counts.min(),
actual_counts.max()], 'r--')
    plt.xlabel('Actual Count')
    plt.ylabel('Predicted Count')
    plt.title('Actual vs Predicted Fish Seed Counts')
    plt.show()
```

Running the Script

Ensure that your dataset is correctly organized as per the Directory Structure section. Update the `phase1_root`, `phase2_root`, and `phase2_counts_file` paths accordingly in the `main()` function.

Save the entire implementation in a Python script, e.g., `fish_seed_counter.py`, and run it using:

```
python fish_seed_counter.py
```

Note: Training deep learning models can be computationally intensive. Ensure you have access to a machine with a GPU to expedite the training process.

Explanation

1. **Dataset Handling**

- Phase 1 (`FishSeedDetectionDataset`):
  - Purpose: Train the model to detect individual fish seeds.
  - Initialization (`__init__`):
    - Loads image filenames and associated annotations from the COCO-formatted `annotations.json`.
    - Creates mappings to efficiently retrieve annotations for each image.
  - `__getitem__`:

    - Loads an image and its bounding box annotations.
    - Converts bounding boxes and labels into tensors.
    - Applies any specified transformations (e.g., converting to tensors).
    - Returns the image along with its target annotations.
- Phase 2 (`FishSeedCountingDataset`):
  - Purpose: Provide images along with their corresponding fish seed counts.
  - Initialization (`__init__`):
    - Loads image filenames and their corresponding counts from `counts.csv`.
    - Applies any specified transformations.
  - `__getitem__`:
    - Loads an image and retrieves its associated count.
    - Returns the image and its count.
- Combined Dataset (`CombinedDataset`):
  - Purpose: Facilitate simultaneous training on Phase 1 and Phase 2 data.
  - `__getitem__`:
    - For indices belonging to Phase 1:
      - Returns image, target (bounding boxes), and `None` for count.
    - For indices belonging to Phase 2:
      - Returns image, `None` for target, and count.

2. Model Configuration

- `get_model` Function:
  - Loads a pre-trained Faster R-CNN model with a ResNet-50 backbone.
  - Replaces the classifier head (`box_predictor`) to match the number of desired classes (in this case, 2: background and "fish_seed").
- `FasterRCNNWithCount` Class:
  - Purpose: Extend Faster R-CNN to predict both object detections and image-level counts.
  - Components:
    - Object Detection: Utilizes the faster R-CNN model for detecting fish seeds.
    - Counting Regression Head (`count_regressor`):
      - Takes aggregated backbone features and predicts the total count of fish seeds in the image.

- Forward Pass:
  - Training:
    - Computes detection losses using bounding box annotations.
    - Extracts backbone features, aggregates them, and predicts counts.
    - Computes counting loss (e.g., L1 loss) and combines it with detection losses.
  - Inference:
    - Performs standard object detection without counting.

3. **Training Pipeline**

- Phase 1: Object Detection Only
  - Objective: Train the model to accurately detect and localize individual fish seeds.
  - Process:
    - Use `FishSeedDetectionDataset` to provide images and bounding boxes.
    - Train the Faster R-CNN component of the model.
    - Evaluate using validation data to monitor losses.
- Phase 2: Object Detection + Counting
  - Objective: Fine-tune the model to also predict the total count of fish seeds per image.
  - Process:
    - Use `CombinedDataset` to provide both Phase 1 and Phase 2 data.
    - For Phase 1 data: Continue training the detection component.
    - For Phase 2 data: Train both detection and counting components.
    - The counting regression head predicts the total count based on image-level labels.
    - Evaluate counting performance using metrics like MAE and RMSE.

4. **Counting Mechanism**

- Object Detection-Based Counting:
  - For images with bounding boxes (Phase 1), counting is straightforward by counting the number of detected objects.
- Counting with Image-Level Labels (Phase 2):
  - The `FasterRCNNWithCount` model predicts counts directly using the counting regression head.

- The loss combines detection losses with counting losses, allowing the model to learn both tasks simultaneously.

5. **Evaluation Metrics**

- Detection Metrics:
  - Loss Values: Monitor the training and validation losses to ensure the model is learning effectively.
  - Visualization: Visualize bounding boxes and compare predicted counts with ground truth.
  - Advanced Metrics: For comprehensive evaluation, integrate metrics like Mean Average Precision (mAP) using libraries like `pycocotools`.
- Counting Metrics:
  - Mean Absolute Error (MAE): Measures the average absolute difference between predicted counts and actual counts.
  - Root Mean Squared Error (RMSE): Measures the square root of the average squared differences between predicted and actual counts.
  - Visualization: Scatter plots of actual vs. predicted counts provide a visual assessment of counting performance.

6. **Visualization**

- Phase 1:
  - Visualize detections on validation images by drawing bounding boxes around detected fish seeds.
  - Display the corresponding confidence scores.
  - Compare the number of ground truth boxes with the number of predictions exceeding the confidence threshold.
- Phase 2:
  - Visualize detections on Phase 2 images.
  - Compare actual counts with predicted counts.
  - Optionally, generate scatter plots to visualize the correlation between actual and predicted counts.

7. **Advanced Considerations**

- Data Augmentation:
  - Implement data augmentation techniques (e.g., random horizontal flips, rotations, brightness adjustments) to enhance model robustness.
  - Modify the `transform` pipeline to include such augmentations during training phases.

- Handling Class Imbalance:
    - If certain counts are overrepresented, consider techniques like weighted loss functions or oversampling to balance the dataset.
- Advanced Counting Mechanisms:
    - Integrate more sophisticated feature aggregation methods for the counting regression head.
    - Explore alternative architectures tailored for counting tasks, such as density map estimation.
- Model Optimization:
    - For deployment, consider optimizing the model using methods like TorchScript, ONNX, or TensorRT for faster inference times.

Complete Code Implementation

Below is the complete code incorporating the `FasterRCNNWithCount` model and integrating both Phase 1 and Phase 2 training regimes.

```
import os
import json
import pandas as pd
import numpy as np
import torch
import torch.utils.data
from torch.utils.data import DataLoader, Dataset
import torchvision
from torchvision import transforms as T
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from collections import defaultdict
from tqdm import tqdm
import torch.nn as nn

# ---------------------
# Custom Dataset Classes
# ---------------------

class FishSeedDetectionDataset(Dataset):
    def __init__(self, root, transforms=None):
        """
        Args:
            root (str): Root directory containing 'images/' and 'annotations.json'.
            transforms (callable, optional): Optional transform to be applied on a
sample.
        """
        self.root = root
```

```
        self.transforms = transforms

        # Load annotation file
        annotation_file = os.path.join(root, "annotations.json")
        with open(annotation_file) as f:
            self.coco = json.load(f)

        # Create mappings
        self.image_id_map = {img['file_name']: img['id'] for img in
self.coco['images']}
        self.annotations = defaultdict(list)
        for ann in self.coco['annotations']:
            # Associate annotations with file names
            image_info = next((img for img in self.coco['images'] if img['id'] ==
ann['image_id']), None)
            if image_info:
                self.annotations[image_info['file_name']].append(ann)

        # List of image filenames
        self.imgs = list(sorted(os.listdir(os.path.join(root, "images"))))

    def __len__(self):
        return len(self.imgs)

    def __getitem__(self, idx):
        # Load image
        img_path = os.path.join(self.root, "images", self.imgs[idx])
        img = Image.open(img_path).convert("RGB")

        # Load annotations
        ann = self.annotations[self.imgs[idx]]

        boxes = []
        labels = []
        areas = []
        iscrowd = []

        for obj in ann:
            xmin, ymin, width, height = obj['bbox']
            boxes.append([xmin, ymin, xmin + width, ymin + height])
            labels.append(1)  # 'fish_seed' category
            areas.append(obj['area'])
            iscrowd.append(obj['iscrowd'])

        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        labels = torch.as_tensor(labels, dtype=torch.int64)
        areas = torch.as_tensor(areas, dtype=torch.float32)
        iscrowd = torch.as_tensor(iscrowd, dtype=torch.int64)

        image_id = torch.tensor([self.coco['images'][idx]['id']])

        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["image_id"] = image_id
```

```
        target["area"] = areas
        target["iscrowd"] = iscrowd

        if self.transforms:
            img = self.transforms(img)

        return img, target

class FishSeedCountingDataset(Dataset):
    def __init__(self, root, counts_file, transforms=None):
        """
        Args:
            root (str): Root directory containing 'images/'.
            counts_file (str): Path to 'counts.csv'.
            transforms (callable, optional): Optional transform to be applied on a
sample.
        """
        self.root = root
        self.transforms = transforms
        self.counts_df = pd.read_csv(counts_file)
        self.imgs = list(self.counts_df['filename'])
        self.counts = list(self.counts_df['count'])

    def __len__(self):
        return len(self.imgs)

    def __getitem__(self, idx):
        # Load image
        img_path = os.path.join(self.root, "images", self.imgs[idx])
        img = Image.open(img_path).convert("RGB")

        count = self.counts[idx]

        if self.transforms:
            img = self.transforms(img)

        return img, count

# ---------------------
# Combined Dataset
# ---------------------

class CombinedDataset(Dataset):
    def __init__(self, phase1_dataset, phase2_dataset):
        """
        Args:
            phase1_dataset (Dataset): Dataset for Phase 1 (Detection).
            phase2_dataset (Dataset): Dataset for Phase 2 (Counting).
        """
        self.phase1_dataset = phase1_dataset
        self.phase2_dataset = phase2_dataset

    def __len__(self):
        return len(self.phase1_dataset) + len(self.phase2_dataset)
```

```
    def __getitem__(self, idx):
        if idx < len(self.phase1_dataset):
            img, target = self.phase1_dataset[idx]
            return img, target, None  # No count label
        else:
            img, count = self.phase2_dataset[idx - len(self.phase1_dataset)]
            return img, None, count  # No target annotations

# ---------------------
# Model Definition
# ---------------------

class FasterRCNNWithCount(nn.Module):
    def __init__(self, num_classes):
        super(FasterRCNNWithCount, self).__init__()
        # Initialize Faster R-CNN
        self.faster_rcnn = get_model(num_classes)
        self.num_classes = num_classes

        # Add a regression head for counting
        # We'll use average pooling on the backbone features and then a fully
connected layer
        # Alternatively, more sophisticated feature aggregation can be used
        # Adjust the input features based on the backbone
        self.count_regressor = nn.Sequential(
            nn.Linear(2048, 512),  # Assuming ResNet-50 backbone output channels
            nn.ReLU(),
            nn.Linear(512, 1)  # Predicting the count
        )

    def forward(self, images, targets=None, counts=None):
        """
        Args:
            images: List of images tensors.
            targets: List of target dictionaries (for detection).
            counts: List or tensor of counts (for counting).
        Returns:
            If training:
                Combined loss dictionary
            Else:
                Detections
        """
        if self.training:
            # Detection losses
            loss_dict = self.faster_rcnn(images, targets)

            # Counting losses
            if counts is not None:
                # Extract features from the backbone
                # For Faster R-CNN, the backbone is a ResNet-50 by default
                # We'll use the output of the backbone before the RPN
                # Access backbone features
                features = self.faster_rcnn.backbone(images.tensors)
                # Typically, for ResNet backbones, features['0'] is the top layer
                backbone_features = features['0']  # Shape: [batch_size, 2048, H, W]
```

```
                # Global Average Pooling
                pooled_features =
torch.nn.functional.adaptive_avg_pool2d(backbone_features, (1, 1))
                pooled_features = pooled_features.view(pooled_features.size(0), -1)
# [batch_size, 2048]

                # Regression Head to predict counts
                count_preds = self.count_regressor(pooled_features).squeeze(1)  #
[batch_size]

                # Convert counts to tensor
                count_targets = torch.tensor(counts,
dtype=torch.float32).to(images.tensors.device)

                # Compute L1 Loss for counting
                count_loss = nn.L1Loss()(count_preds, count_targets)

                # Combine losses
                total_loss = sum(loss for loss in loss_dict.values()) + count_loss
                loss_dict['loss_count'] = count_loss
                loss_dict['total_loss'] = total_loss

                return loss_dict
            else:
                return self.faster_rcnn(images, targets)
        else:
            # Inference
            detections = self.faster_rcnn(images)
            return detections

def get_model(num_classes):
    # Load a pre-trained Faster R-CNN model
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)

    # Get the number of input features for the classifier
    in_features = model.roi_heads.box_predictor.cls_score.in_features

    # Replace the pre-trained head with a new one
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    return model
```

Training and Evaluation Functions

```
def train_phase1(model, optimizer, data_loader, device, epoch):
    model.train()
    running_loss = 0.0
    for images, targets in tqdm(data_loader, desc=f"Epoch {epoch} Phase 1 Training"):
        images = list(image.to(device) for image in images)
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        # Forward pass
        loss_dict = model(images, targets)
```

```
        # Total loss is the sum of individual losses
        losses = sum(loss for loss in loss_dict.values())

        # Backward pass and optimization
        optimizer.zero_grad()
        losses.backward()
        optimizer.step()

        running_loss += losses.item()

    avg_loss = running_loss / len(data_loader)
    print(f"Epoch {epoch} Phase 1 - Average Loss: {avg_loss:.4f}")

def evaluate_phase1(model, data_loader, device):
    model.eval()
    total_loss = 0.0
    with torch.no_grad():
        for images, targets in tqdm(data_loader, desc="Phase 1 Evaluation"):
            images = list(image.to(device) for image in images)
            targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

            # Forward pass
            loss_dict = model(images, targets)

            # Total loss
            losses = sum(loss for loss in loss_dict.values())
            total_loss += losses.item()

    avg_loss = total_loss / len(data_loader)
    print(f"Phase 1 Evaluation - Average Loss: {avg_loss:.4f}")
    return avg_loss

def train_phase2(model, optimizer, data_loader, device, epoch):
    model.train()
    running_loss = 0.0
    for images, targets, counts in tqdm(data_loader, desc=f"Epoch {epoch} Phase 2
Training"):
        images = list(image.to(device) for image in images)
        batch_targets = []
        batch_counts = []
        for tgt, cnt in zip(targets, counts):
            if tgt is not None:
                batch_targets.append(tgt)
            if cnt is not None:
                batch_counts.append(cnt)

        if len(batch_targets) > 0:
            # Forward pass with targets and counts
            loss_dict = model(images, targets=batch_targets, counts=batch_counts)
            losses = loss_dict['total_loss']

            # Backward pass and optimization
            optimizer.zero_grad()
            losses.backward()
            optimizer.step()
```

```
            running_loss += losses.item()
        else:
            continue  # Skip if no targets and counts

    avg_loss = running_loss / len(data_loader)
    print(f"Epoch {epoch} Phase 2 - Average Loss: {avg_loss:.4f}")

def evaluate_counting(model, data_loader, device, threshold=0.5):
    model.eval()
    predicted_counts = []
    actual_counts = []
    with torch.no_grad():
        for images, counts in tqdm(data_loader, desc="Phase 2 Counting Evaluation"):
            images = list(image.to(device) for image in images)
            outputs = model.faster_rcnn(images)

            for output, count in zip(outputs, counts):
                detected = 0
                for score, label in zip(output['scores'], output['labels']):
                    if score > threshold and label == 1:
                        detected += 1
                predicted_counts.append(detected)
                actual_counts.append(count)

    predicted_counts = np.array(predicted_counts)
    actual_counts = np.array(actual_counts)

    mae = np.mean(np.abs(predicted_counts - actual_counts))
    rmse = np.sqrt(np.mean((predicted_counts - actual_counts) ** 2))

    print(f"Phase 2 Counting - MAE: {mae:.2f}, RMSE: {rmse:.2f}")
    return mae, rmse
```

Visualization Functions

```
def visualize_predictions_phase1(model, dataset, device, threshold=0.5, 
num_samples=5):
    model.eval()
    for i in range(num_samples):
        img, target = dataset[i]
        with torch.no_grad():
            prediction = model([img.to(device)])[0]

        img_np = img.permute(1, 2, 0).cpu().numpy()

        fig, ax = plt.subplots(1)
        ax.imshow(img_np)
        true_count = len(target['boxes'])
        pred_count = 0
        for box, score, label in zip(prediction['boxes'], prediction['scores'], 
prediction['labels']):
            if score > threshold and label == 1:
                pred_count += 1
                xmin, ymin, xmax, ymax = box.cpu()
```

```
                rect = patches.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
                                         linewidth=2, edgecolor='r',
facecolor='none')
                ax.add_patch(rect)
                ax.text(xmin, ymin, f"{score:.2f}", bbox=dict(facecolor='yellow',
alpha=0.5))

        plt.title(f"Ground Truth: {true_count}, Predicted: {pred_count}")
        plt.axis('off')
        plt.show()

def visualize_predictions_phase2(model, dataset, device, threshold=0.5,
num_samples=5):
    model.faster_rcnn.eval()
    for i in range(num_samples):
        img, count = dataset[i]
        img_input = img.unsqueeze(0).to(device)
        with torch.no_grad():
            prediction = model.faster_rcnn(img_input)[0]

        img_np = img.permute(1, 2, 0).cpu().numpy()

        fig, ax = plt.subplots(1)
        ax.imshow(img_np)
        pred_count = 0
        for box, score, label in zip(prediction['boxes'], prediction['scores'],
prediction['labels']):
            if score > threshold and label == 1:
                pred_count += 1
                xmin, ymin, xmax, ymax = box.cpu()
                rect = patches.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
                                         linewidth=2, edgecolor='r',
facecolor='none')
                ax.add_patch(rect)
                ax.text(xmin, ymin, f"{score:.2f}", bbox=dict(facecolor='yellow',
alpha=0.5))

        plt.title(f"Actual: {count}, Predicted: {pred_count}")
        plt.axis('off')
        plt.show()
```

Main Training and Inference Pipeline

```
def main():
    # Paths to dataset phases
    phase1_root = 'dataset/phase1'  # Replace with your actual path
    phase2_root = 'dataset/phase2'  # Replace with your actual path
    phase2_counts_file = os.path.join(phase2_root, "counts.csv")

    # Define transforms
    transform = T.Compose([
        T.ToTensor(),
    ])

    # Create datasets
```

```
    dataset_phase1 = FishSeedDetectionDataset(phase1_root, transforms=transform)
    dataset_phase2 = FishSeedCountingDataset(phase2_root, phase2_counts_file,
transforms=transform)

    # Split Phase 1 into training and validation
    val_split = 0.1
    num_val = int(len(dataset_phase1) * val_split)
    num_train = len(dataset_phase1) - num_val
    dataset_phase1_train, dataset_phase1_val =
torch.utils.data.random_split(dataset_phase1, [num_train, num_val])

    # DataLoaders for Phase 1
    data_loader_phase1_train = DataLoader(dataset_phase1_train, batch_size=4,
shuffle=True, num_workers=4,
                                          collate_fn=lambda x: tuple(zip(*x)))
    data_loader_phase1_val = DataLoader(dataset_phase1_val, batch_size=4,
shuffle=False, num_workers=4,
                                        collate_fn=lambda x: tuple(zip(*x)))

    # DataLoader for Phase 2
    # For Phase 2, we need to provide images with counts. We'll create a combined
dataset.
    # To handle both datasets, we'll use a custom collate_fn
    combined_dataset = CombinedDataset(dataset_phase1_train, dataset_phase2)
    data_loader_combined = DataLoader(combined_dataset, batch_size=4, shuffle=True,
num_workers=4,
                                      collate_fn=lambda x: tuple(zip(*x)))

    # Define the device
    device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
    print(f"Using device: {device}")

    # Define the number of classes (background + fish_seed)
    num_classes = 2

    # Initialize the model
    model = FasterRCNNWithCount(num_classes)
    model.to(device)

    # Define optimizer (commonly SGD for Faster R-CNN)
    params = [p for p in model.parameters() if p.requires_grad]
    optimizer = torch.optim.SGD(params, lr=0.005,
                                momentum=0.9, weight_decay=0.0005)

    # Learning rate scheduler
    lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                                   step_size=3,
                                                   gamma=0.1)

    num_epochs_phase1 = 5
    num_epochs_phase2 = 10

    # Phase 1: Train object detection
    print("Starting Phase 1 Training (Object Detection)...")
```

```
    for epoch in range(1, num_epochs_phase1 + 1):
        train_phase1(model.faster_rcnn, optimizer, data_loader_phase1_train, device,
epoch)
        evaluate_phase1(model.faster_rcnn, data_loader_phase1_val, device)
        lr_scheduler.step()

    print("Phase 1 Training Completed.\n")

    # Save the Phase 1 model
    torch.save(model.state_dict(), "fasterrcnn_fish_seed_phase1.pth")
    print("Phase 1 Model saved as fasterrcnn_fish_seed_phase1.pth")

    # Visualization for Phase 1
    visualize_predictions_phase1(model.faster_rcnn, dataset_phase1_val, device,
threshold=0.5, num_samples=3)

    # Phase 2: Fine-tune with counting
    print("Starting Phase 2 Training (Object Detection + Counting)...")
    for epoch in range(1, num_epochs_phase2 + 1):
        train_phase2(model, optimizer, data_loader_combined, device, epoch)
        lr_scheduler.step()

    print("Phase 2 Training Completed.\n")

    # Save the Phase 2 model
    torch.save(model.state_dict(), "fasterrcnn_fish_seed_phase2.pth")
    print("Phase 2 Model saved as fasterrcnn_fish_seed_phase2.pth")

    # Evaluation for Phase 2
    mae, rmse = evaluate_counting(model, DataLoader(dataset_phase2, batch_size=4,
shuffle=False, num_workers=4), device, threshold=0.5)

    # Visualization for Phase 2
    visualize_predictions_phase2(model, dataset_phase2, device, threshold=0.5,
num_samples=3)

    # Optionally, plot predicted vs actual counts
    predicted_counts = []
    actual_counts = []
    model.eval()
    with torch.no_grad():
        for i in range(len(dataset_phase2)):
            img, count = dataset_phase2[i]
            img_input = img.unsqueeze(0).to(device)
            prediction = model.faster_rcnn(img_input)[0]
            pred_count = sum((score > 0.5 and label == 1) for score, label in
zip(prediction['scores'], prediction['labels']))
            predicted_counts.append(pred_count)
            actual_counts.append(count)

    plt.figure(figsize=(8,6))
    plt.scatter(actual_counts, predicted_counts, alpha=0.6)
    plt.plot([actual_counts.min(), actual_counts.max()], [actual_counts.min(),
actual_counts.max()], 'r--')
    plt.xlabel('Actual Count')
```

```
    plt.ylabel('Predicted Count')
    plt.title('Actual vs Predicted Fish Seed Counts')
    plt.show()
```

Running the Script

1. Ensure Dataset Organization:
   - Verify that your dataset is organized as per the Directory Structure section.
   - Update the `phase1_root`, `phase2_root`, and `phase2_counts_file` paths in the `main()` function if they differ.
2. Execute the Script:
   - Save the complete implementation in a Python script, e.g., `fish_seed_counter.py`.
   - Run the script using:

   ```
   python fish_seed_counter.py
   ```

Note: Training deep learning models can be computationally intensive. Ensure you have access to a machine with a GPU to expedite the training process.

Final Notes

- Data Quality and Quantity:
  - The performance of both detection and counting heavily depends on the quality and diversity of your dataset.
  - Ensure that Phase 1 includes varied perspectives of single fish seeds to help the model generalize well in Phase 2.
- Confidence Threshold (`threshold=0.5`):
  - Adjust this value based on validation performance to optimize counting accuracy.
  - A higher threshold may reduce false positives but miss some detections, while a lower threshold may increase detections but include more false positives.
- Model Saving:
  - The script saves two versions of the model:
    - Phase 1: `fasterrcnn_fish_seed_phase1.pth`
    - Phase 2: `fasterrcnn_fish_seed_phase2.pth`
  - Use Phase 2 model for the most accurate counting as it incorporates counting capabilities.
- Advanced Counting Mechanisms:
  - The current counting approach leverages the detection outputs. For more sophisticated counting, consider integrating density maps or other counting-specific architectures.

- Evaluation Metrics:
    - o For object detection, consider integrating more comprehensive metrics like Mean Average Precision (mAP) using libraries such as `pycocotools`.
    - o For counting, metrics like MAE and RMSE provide insights into the counting accuracy.
- Optimization and Deployment:
    - o For deployment purposes, consider optimizing the model using techniques like TorchScript or exporting to ONNX format for compatibility with various platforms.

By following this implementation, you can train a robust model capable of detecting and counting fish seeds in images, even in scenarios where bounding box annotations are unavailable for multi-seed images.